IMPLEMENTATION OF LEAST FREQUENTLY USED (LFU) REPLACEMENT POLICY AND CACHE PREFETCHER

Group Members:
Aditya Dhananjay Singh - 22CS02001
Kumar Snehal - 22CS02009
Atharva Atul Penkar - 22CS02011

Objectives:

1. Implementation of Least Frequently Used Replacement Policy.
2. Implementation of Next Line and Stride Prefetching Algorithm.
3. Building Complex Benchmarks to analyse variations in performance of Level 1 Caches.
4. Analysing change in performance with respect to associativity of a set.

Changes to Simple Scalar:

LFU Cache Replacement Policy

This document describes the implementation of the Least Frequently Used (LFU) cache replacement policy in the SimpleScalar cache module (cache.c). Our primary goal was to improve the predictability of cache evictions by tracking the long-term history of block usage.

Technical Implementation

Our implementation required modifying the core data structures and logic within cache.c:

1. **Block Metadata Expansion:** We introduced a new field, **freq_count** (unsigned integer), into the struct cache_blk_t to persistently store the access history for every cache line.
2. **Counting Mechanism:**
    - **Initialization:** Upon a cache miss resulting in a block fill, we initialized the victim block's freq_count to **1**.
    - **Updating:** On every subsequent cache hit, we implemented logic in the cache_hit and cache_fast_hit handlers to atomically increment the block's freq_count.
3. **Victim Selection:** The core LFU logic was integrated into the cache miss handler's replacement phase (switch (cp->policy)). When LFU was active, our routine iteratively scanned all blocks within the required cache set (using the way_head pointer to traverse the set's linked list) to identify the block holding the absolute minimum freq_count. This block was then selected as the replacement victim.

LFU Working Principle

The critical benefit of our LFU implementation is its ability to filter transient noise:

- **Protection of Hot Data:** By relying on cumulative frequency rather than just recency, blocks accessed repeatedly over many cycles were assigned a high freq_count, effectively shielding them from eviction by temporary data floods (unlike LRU, which might evict them if the cycle is long enough).
- **Capacity Miss Handling:** When subjected to capacity pressure—as seen in our multi-stream benchmarks—LFU ensured that if an eviction was necessary, the choice was the block with the least proven utility across the entire runtime.

This methodical integration ensures that our LFU policy accurately prioritizes high-utility data and provides a strong baseline for evaluating cache performance against classic policies like LRU and FIFO.

Prefetcher

The prefetcher module in the SimpleScalar simulator was extended to support multiple prefetching policies:

- **None**: Baseline cache with no prefetching.
- **Next-Line Prefetcher**: Issues a prefetch for the immediately next cache block after every access.
- **Stride Prefetcher**: Detects regular strides in memory access patterns per stream and prefetches blocks ahead based on detected stride.

Technical Implementation

Our implementation required modifying the core data structures and logic within cache.c, as well as making a change in sim-cache.c (to incorporate the new input).

1. **Command:** Added the -prefetcher command-line option in sim-cache to select the prefetcher policy dynamically.
2. **Block Metadata Expansion:** Extended struct cache_t to enum prefetcher_policy prefetcher a stride prefetcher state table.
3. **Prefetching Mechanism:**
    - Whenever cache_access() function is called the cache_prefetch_logic() is called in case **cache misses** and **slow hits** (not on fast hits).
    - In cache_prefetch_logic() according to the prefetching algorithm the next block is fetched according to the prefetching algorithm by calling the cache_issue_prefetch() function.
    - The cache_issue_prefetch() works similar to the normal cache_access function but does not incur any miss penalty or time delay, if the block already exists nothing happens otherwise a block is victimised according to the replacement algorithm.

Prefetching Working Principle

How Is the Prefetch Address Determined?

1. **Next-Line Prefetcher:**
    a. The predictor always assumes the program accesses memory sequentially (like reading pages in order).
    b. When a block at address XX is accessed, the next-line prefetcher issues a request for X+block_sizeX+block_size—the very next cache line, no matter the actual access pattern.
    c. **Example**: If block size = 64 bytes and you access 0x1000, it'll prefetch 0x1040 next, even if the next access isn't there.
2. **Stride Prefetcher:**
    a. The stride prefetcher maintains a small table for tracking how far apart your memory accesses are.
    b. For each table entry, it records:
        i. Last block accessed.
        ii. Last detected stride (difference between consecutive accesses).
        iii. A confidence counter indicating how stable this stride is.
    c. **Example**: If accesses go 0x1000, 0x1040, 0x1080 (stride=64B), once the stride is detected, future accesses will trigger prefetches for the next stride (e.g., prefetch 0x10C0 when at 0x1080).

Why Does This Work?

- **Next-Line:**
    - Works best for truly sequential access (array traversal, streaming). It's simple and aggressive, but wasteful on non-sequential patterns because it may prefetch many useless blocks.
- **Stride:**
    - Adapts to regular patterns (strided array walks), such as jumping by multiples of the block size, column accesses in matrices, or structures with regular gaps. It can handle patterns where the next access isn't the next line, but a constant distance away.
    - By learning stride patterns, the prefetcher brings data into the cache before it's needed, hiding memory latency and boosting performance for such workloads.

Key Design Choices and Logic

1. **Indexing Without PC (Stride Only):** To keep things simple and practical, stride entries are indexed by a hash of block address, using an XOR-folding trick to reduce aliasing and make pattern tracking more reliable without needing the program counter.
2. **Confidence Management (Stride Only):** The stride prefetcher tracks a confidence score for each stride. Matching strides boost confidence; a changed stride resets to 1 for fast relearning.

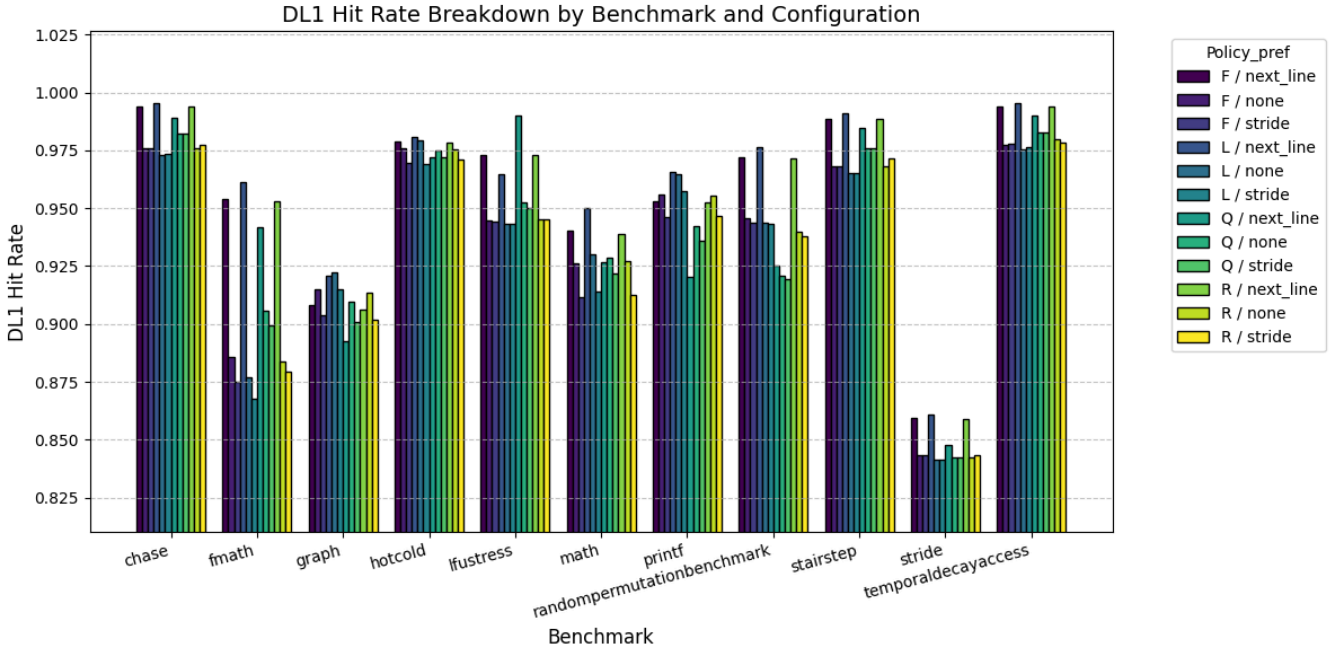
3. **Prefetch Threshold:** As soon as confidence crosses 1 (indicating a repeat pattern), stride prefetches begin to avoid waiting for long runs of identical strides.
4. **Non-blocking:** Both prefetchers issue non-blocking requests—fetched data fills the cache in the background and doesn't stall the processor.

Impact

- **Next-Line Prefetcher:** Improves hit rates for sequential workloads, may waste bandwidth for irregular accesses.
- **Stride Prefetcher:** Learns regular gap patterns and prefetches them, boosting cache efficiency for strided, matrix, or structured memory patterns. Reduces latency and cache misses in stride-heavy cases, with less wasted bandwidth for non-sequential workloads.

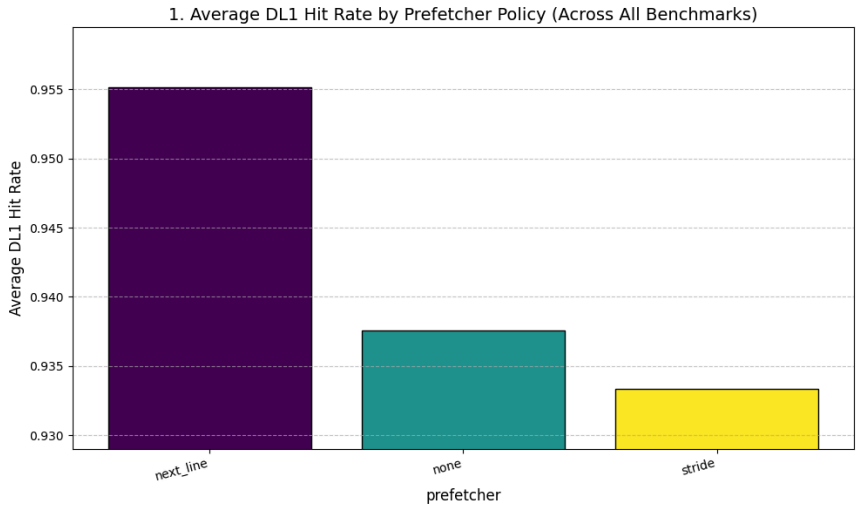
Analysis:

Data Level 1 Cache Data:



We are using the Data Cache since we know that the Instruction will naturally be similar due to the codes being stored together.
There are a lot of metrics here which cover all the replacement policies, and prefetching algorithms with the benchmarks we found or created and used.
We will be covering specific data points to perform our analysis.

Prefetching Policies:



We have taken the average of DL1 hit rates across all the benchmarks and replacement policies.

The graph indicates that the **next line prefetcher** delivers the best performance. This superior result is attributed to the spatial locality present in the arrays within most of our benchmark codes, which the next line prefetcher effectively exploits.

Following this, the **prefetcher with no specific settings** shows intermediate performance.

Conversely, the **stride prefetcher** performs poorly because the benchmarks lack the large strides required for it to be effective.

Replacement Policy Breakdown:

LRU (Least Recently Used) LRU provides the best prediction by perfectly exploiting temporal locality. It reliably evicts the block that has been unused for the longest time.

LFU (Least Frequently Used) LFU tracks cumulative access count, protecting the blocks used

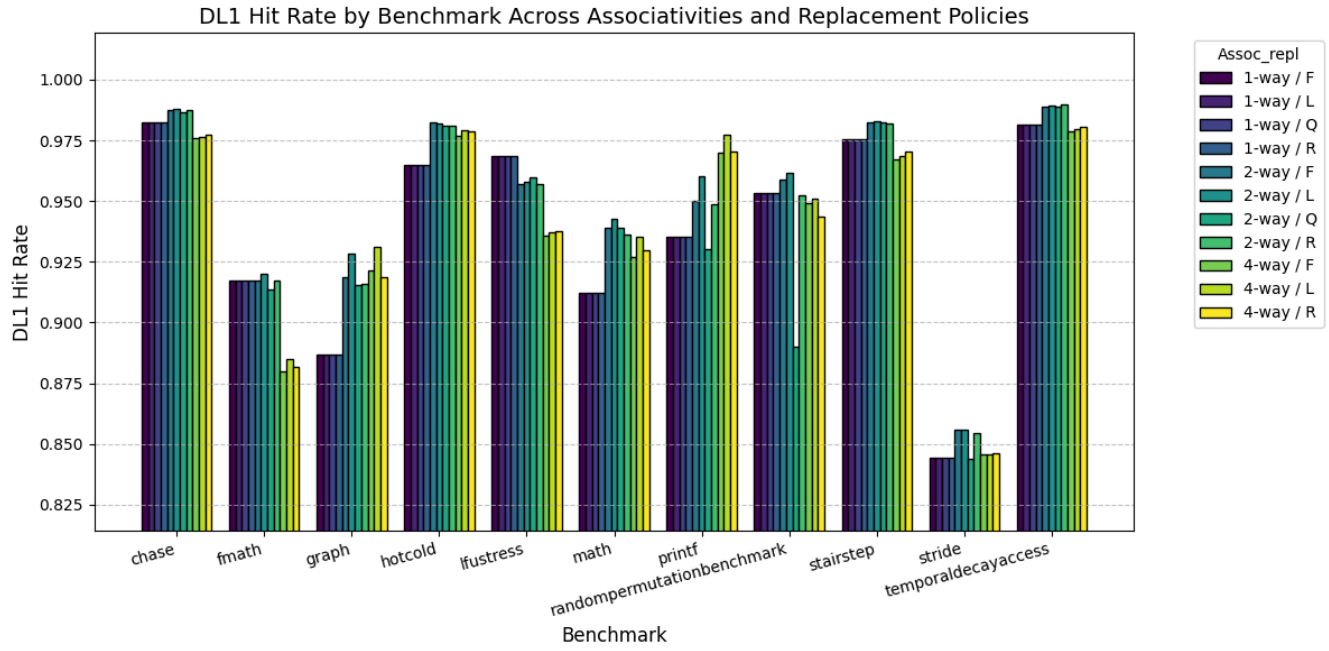
most over the long term. It can fail if a critical, non-repeating instruction block is evicted because its count is low.

FIFO (First-In, First-Out) FIFO evicts the oldest block in the cache, ignoring how recently or often it has been used. This complete disregard for temporal locality leads to poor performance in cyclic workloads.

Random Random replacement offers no predictive ability and serves as the performance baseline. It will perform arbitrary evictions, resulting in the highest unnecessary miss rate.

LRU replacement policy performs the best out of all.

Associativities effect along with Replacement Policies:



1-way (Direct Mapped): Lowest performance due to maximum **Conflict Misses** (blocks always fight for the same slot). Check performance decrease in .

2-way & 4-way: Significantly reduces conflict misses, with 2-way giving the largest jump in hit rate. **4-way** offers minimal extra gain but approaches ideal mapping by reducing competition to only Capacity Misses.